Survey:
Certified Compilation/Certifying Compilation

rei@506

Introduction

Motivations

Survey about the formalization of the safety of Java $\rightarrow$ PCC (?) $\rightarrow$ LF $\rightarrow$ Twelf $\rightarrow$ Coq $\rightarrow$ Type Theory, Semantics $\rightarrow$ PE $\rightarrow$ Certified compilers $\rightarrow$ PCC (!)

Outline

1. Certified compilation: an experiment
2. Certifying compilation: an experiment

A certified compiler for an imperative language ([Ber98])

Introduction

Mechanical proof of correctness of a compiler for an imperative language to an assembly language.

Coq as a Specification Environment

Specification aspect:

Abstract syntax specification $\leftrightarrow$ Coq inductive types

Semantic specification $\leftrightarrow$ Coq inductive predicates

Proof aspect:

Coq provides specialized proof tactics for manipulating inductive constructs (automatic generation of structural induction principles [PMW01])

Source Language — Syntax

Mutually inductive types $PROGRAM$, $INST$, $DECLS$, $DECL$, EXP, ID, VAL, $TYPE$ that represent the core of most imperative languages. For instance:

$INST$ has 4 constructors:

- $assign : ID \to EXP \to INST$
- $if : EXP \to INST \to INST \to INST$
- $sequence : list(LIST) \to INST$
- $while : EXP \to INST \to INST$

$DECL$ has 1 constructor:

- $decl : ID \to TYPE \to VAL \to DECL$

$PROGRAM$ is a list of instructions, $DECLS$ is a list of declaration

Source Language — Dynamic Semantics

Inductive predicates *exec*, *eval* that represent the semantics of most imperative languages:

$$\forall e : EXP.\forall d, d' : DECLS.\forall I : ID.\forall v : VAL.$$
$$eval(d, e, v) \rightarrow update(d, I, v, d') \rightarrow exec(d, assign(I, e), d')$$
$$\forall i_1, i_2 : INST.\forall e : EXP.\forall d, d' : DECLS.$$
$$eval(d, e, true) \rightarrow exec(d, i_1, d') \rightarrow exec(d, if(e, i_1, i_2), d')$$
$$\forall i_1, i_2 : INST.\forall e : EXP.\forall d, d' : DECLS.$$
$$eval(d, e, false) \rightarrow exec(d, i_2, d') \rightarrow exec(d, if(e, i_1, i_2), d')$$
$$\forall i : INST.\forall e : EXP.\forall d : DECLS.$$
$$eval(d, e, false) \rightarrow exec(d, while(e, i), d)$$
$$\forall i : INST.\forall e : EXP.\forall d, d', d'' : DECLS.$$
$$eval(d, e, true) \rightarrow exec(d, i, d') \rightarrow exec(d', while(e, i), d'') \rightarrow exec(d, while(e, i), d'')$$
$$\forall i_1 : INST.\forall i_2 : list(INST).\forall d, d', d'' : DECLS.$$
$$exec(d, i_1, d') \rightarrow exec(d', sequence(i_2), d'') \rightarrow exec(d, sequence(i_1.i_2), d'')$$
$$\forall d : DECLS.exec(d, sequence([]), d)$$

Target Language — Syntax

Inductive type *Assembly* that represents the instruction set of a machine with one register, a stack, and a memory:

Constructors:

- *label*, *goto*, *branch_if_0*
- *load*, *store*
- *push*, *pop*
- *add*, *sub*, *not*, *eq*, *gt*
- *immediate*

Target Language — Dynamic Semantics

Inductive predicate $execute : list(Assembly) \rightarrow exec_state \rightarrow list(Assembly) \rightarrow exec_state \rightarrow Prop$ that represents the semantics of the assembly language:

2 Constructors:[a]

- $execute_end : \forall pg : list(Assembly).\forall s : exec_state.execute(pg, s, [], s)$
- $execute_rec :$

$$\forall pg, pg', pg'' : list(Assembly).\forall i : Assembly.$$
$$\forall m, s, s' : list(nat).\forall r, r' : nat.\forall result : exec_state.$$
$$new_pc(pg, pg', r, i) = pg'' \rightarrow new_stack(r, s, i) = s' \rightarrow$$
$$new_register(r, m, s, i) = r' \rightarrow$$
$$execute(pg, (new_mem(r, m, i), s', r'), pg'', result) \rightarrow$$
$$execute(pg, s, i.pg', result).$$

[a] new_stack, $new_register$, new_mem, new_pc are functions

The Compiler

A set of structural recursive functions:

- *compile_val* for booleans and integers
- *compile_bin* for binary operations
- *compile_exp*(E) for expressions
- *compile_inst*(pg, n) for an instruction
- *compile_seq* for a sequence of instructions

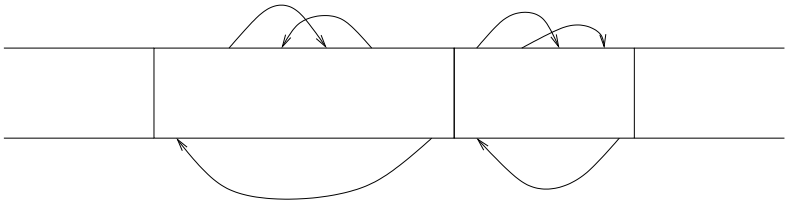
For instance:[a]

$$[[if(E, I_1, I_2)]]_n$$
$$= [[E]].branch_if_0(n).[[I_1]].goto(n+1).label(n).[[I_2]].label(n+1)$$

[a] $[[\cdot]]$ is a shortcut for the compilation function

Fragmentation Theorems

Self-contained programs:



Decompose Theorem: "if the execution of the concatenation of a self-contained program and an arbitrary program terminates, then we can first execute the first program with itself as context, and then execute the second program with the whole program as context."

Other Properties

Type-checking: A collection of inductive predicates *check_decls*, *check_exp*, *check_inst*,

Type-compatibility: two declaration blocks are if they declare the same variables

$\Rightarrow$ Subject reduction theorems: For instance, "the initial and final execution blocks when executing an instruction are *types_compatible*."

Generalized induction principle: [a]

$$\forall pg : list(Assembly).\forall P.\forall s, s''.$$
$$(\forall pg'' : list(Assembly).\forall s'.\forall pg'.execute'(pg, s, pg', s', pg'') \rightarrow$$
$$execute(pg, s', pg'', s'') \rightarrow P(s', pg'', s'')) \rightarrow$$
$$\forall s_0, s_1.\forall pg_0 : list(Assembly).P(s_0, pg_0, s_1)$$

[a] stems from $(\forall n''.(\forall m'.m' < n'' \rightarrow P(m')) \rightarrow P(n'')) \rightarrow \forall n_0.P(n_0)$

Correctness Proof

1st Part: Completeness

What the source program does is also done by the compiled program:

$$\forall i : INST.\forall d, d' : DECLS.\forall s : exec_state.$$
$$exec(d, i, d') \rightarrow check_inst(d, i) \rightarrow check_decls(d) \rightarrow coherent(d, s) \rightarrow$$
$$\forall n : nat.\exists s' : exec_result.execute([[i]]_n, s, [[i]]_n, s') \wedge coherent(d', s')$$

Induction on the structure of the derivation of *exec*. Inversion on the predicate *check_inst*. Subject reduction properties.

Correctness Proof

2nd Part: Soundness

What the compiled program does is also done by the source program:

$$\forall d : DECLS. \forall i : INST. \forall s, s' : exec_state. \forall n : nat.$$
$$check_inst(d, i) \rightarrow execute([[i]]_n, s, [[i]]_n, s') \rightarrow check_decls(d) \rightarrow$$
$$coherent(d, s) \rightarrow$$
$$\exists d' : DECLS. exec(d, i, d') \wedge coherent(d', s')$$

$[[i]]$ lacks structure $\Rightarrow$ Induction on the structure of the derivation of *check_inst*.

Correctness Proof

2nd Part: Soundness — Reformulated (1/2)

$\forall d : DECLS. \forall i : INST. check_inst(d, i) \rightarrow \forall s, s' : exec_state.$

$\forall n : nat. execute([[i]]_n, s, [[i]]_n, s') \rightarrow$

$\forall d' : DECLS. types_compatible(d, d') \rightarrow check_decls(d') \rightarrow coherent(d', s) \rightarrow$

$\exists d'' : DECLS. exec(d', i, d'') \wedge coherent(d'', s') \wedge types_compatible(d, d'')$

Case $i = I_1; I_2$:
$[[i]] = [[I_1]]; [[I_2]]$. Goal: the existence of a final declaration block for the execution of i. Fragmentation theorems $\Rightarrow$ there is some s_1 such that $execute([[I_1]], s, [[I_2]], s_1)$ and $execute([[I_2]], s_1, [[I_2]], s')$. Induction hypothesis $\Rightarrow$ $exec(d', I_1, d_1)$ where d_1 is well-typed because of subject reduction properties. Induction hypothesis $\Rightarrow$ $exec(d_1, I_2, d'')$ where d'' has the expected properties.

Correctness Proof

2nd Part: Soundness — Reformulated (2/2)

$\forall d : DECLS. \forall i : INST. check_inst(d, i) \rightarrow \forall s, s' : exec_state.$

$\forall n : nat. execute([[i]]_n, s, [[i]]_n, s') \rightarrow$

$\forall d' : DECLS. types_compatible(d, d') \rightarrow check_decls(d') \rightarrow coherent(d', s) \rightarrow$

$\exists d'' : DECLS. exec(d', i, d'') \wedge coherent(d'', s') \wedge types_compatible(d, d'')$

Case $i = while(E, I)$:
$[[i]] = label(n).[[E]].branch_if_0(n+1).[[I]].goto(n).label(n+1)$. Goal:
the existence of a final declaration block for the execution of i.
Soundness property for the expression compiler $\Rightarrow eval(d', E, true)$.
Induction hypothesis $\Rightarrow exec(d', I, d_1)$ where d_1 is wellp-typed because of
subject reduction properties. Generalized induction principle $\Rightarrow$
$exec(d_1, while(E, I), d'')$ where d'' has the right properties.

A certified compiler for an imperative language

Proof script file

realization: 3 months (by an intensive Coq user, the language specification stemming from a previous work)
length: 8,000 lines
execution time: 15 minutes (PI, 333MHz, 128MB)
extraction properties: no manual rewriting is needed

Related work

[BF95] (proof of constant propagation and code simplication), [Bou95] (proof of correctness of the compilation of Mini-ML to the CAM VM), [Hat95] (proof of correctness of partial evaluation for a λ-calculus), [Ber01] (formalization of a JVML verifier for initialization), [PMW01] (specification of an imperative language with proofs)

A certified compiler for an imperative languag

Conclusion

Inconvenients:

- Proof is difficult.
- Changes in the compiler require the proof to be updated.
- The trickier optimizations are, the more difficult the proof is.
- The certification does not survive code mobility.

The original goal was ensuring that the output of the compiler is correct. Solutions:

- Restrict the proof of equivalence 'compiler input'/'compiler output' to a proof on the compiler output only.
- Restrict the notion of correctness to safety.
- Staging: Off-line proof generation/On-line proof checking.

Proof-Carrying Code ([Nec97])

Introduction

Problem: How can a system determine whether it is safe to run a program supplied by an untrusted source?

Problem instance: How can a run-time system determine whether it is safe to run a new primitive function supplied by an untrusted source in binary form?

Case Study: the TIL[a] run-Time System/DEC Alpha assembly language extensions

(1) the compiler designer defines a *safety policy* for extensions,
(2) the code producer produces a piece of (foreign) code and
(3) attaches a *safety proof* that
(4) he has beforehand encoded.

[a] compiler for SML + run-time system written in C

Safety Policy (1/4)

Definition

Formal description of the representation of data +
Calling conventions

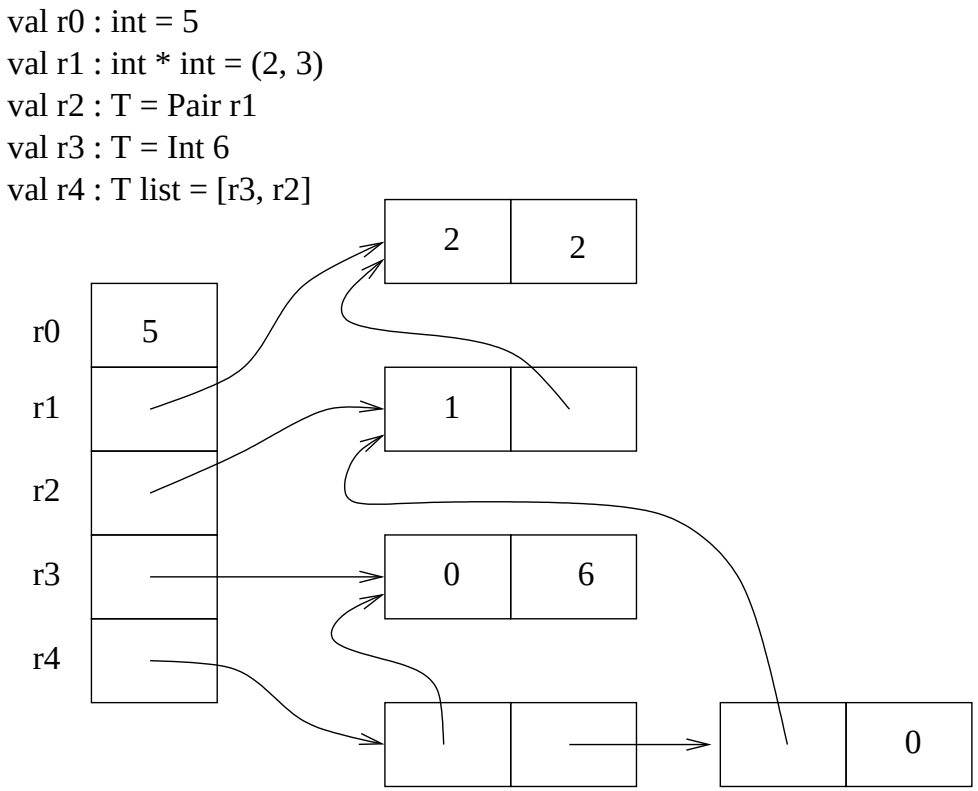
of the TIL compiler & run-time system

Safety Policy (2/4)

Concrete Representation of Data

Concrete representation of data by types:

$\tau ::= \texttt{int} | \tau_1 * \tau_2 | \tau_1 + \tau_2 | \tau\texttt{list} | \texttt{T}$; where $\texttt{T} \equiv \texttt{int} + (\texttt{int} * \texttt{int})$



Safety Policy (3/4)

Formal Representation of Data...

Formal representation of data by a typing judgment:

$$m \vdash e : \tau$$

where e is an expression, m is a memory state, τ is a type:

$$e ::= n | \mathtt{r}_i | \mathtt{sel}(m, e) | e_1 + e_2$$

$$m ::= \mathtt{r}_m | \mathtt{upd}(m, e_1, e_2)$$

$$\tau ::= \mathtt{addr} | \mathtt{ro_addr}$$

Safety Policy (4/4)

...Formal Representation of Data

A set of type inference rules:

$$\frac{m \vdash e : \tau_1 * \tau_2}{m \vdash e : \texttt{addr} \wedge m \vdash e + 4 : \texttt{addr} \wedge m \vdash \texttt{sel}(m, e) : \tau_1 \wedge m \vdash \texttt{sel}(m, e + 4) : \tau_2}$$

$$\frac{m \vdash e : \tau_1 + \tau_2}{\begin{array}{c} m \vdash e : \texttt{addr} \wedge m \vdash e + 4 : \texttt{addr} \wedge \\ \texttt{sel}(m, e) = 0 \supset m \vdash \texttt{sel}(m, e + 4) : \tau_1 \wedge \texttt{sel}(m, e) \neq 0 \supset m \vdash \texttt{sel}(m, e + 4) : \tau_2 \end{array}}$$

$$\frac{m \vdash e : \tau\texttt{list} \quad e \neq 0}{m \vdash e : \texttt{addr} \wedge m \vdash e + 4 : \texttt{addr} \wedge m \vdash \texttt{sel}(m, e) : \tau \wedge m \vdash \texttt{sel}(m, e + 4) : \tau\ \texttt{list}}$$

$$\frac{m \vdash e_1 : \texttt{int} \quad m \vdash e_2 : \texttt{int}}{m \vdash e_1 + e_2 : \texttt{int}} \qquad \frac{}{m \vdash 0 : \texttt{int}}$$

The Foreign Function (1/2) — in SML

```
datatype T = Int of int | Pair of int * int

fun sum (l: T list) =
  let fun foldr f nil a = a
        | foldr f (h::t) a = foldr f t (f (a, h))
  in
    foldr (fn (acc, Int i) => acc + i
            | (acc, Pair (i, j)) => acc + i + j)
          l 0
  end
```

The Foreign Function (2/2) — in ASM

```
0 sum:  INV    r_m ⊢ r_0 : T list % r_0 is l
1       MOV    r_1, 0 % Initialize accumulator
2 L2    INV    r_m ⊢ r_0 : T list ∧r_m ⊢ r1 : int % Loop invariant
3       BEQ    r_0, L14 % Is list empty?
4       LD     r_2, 0(r_0) % Loop head
5       LD     r_0, 4(r_0) % Loop tail
6       LD     r_3, 0(r_2) % Load constructor
7       LD     r_2, 4(r_2) % Load data
8       BEQ    r_3, L12 % Is data an integer?
9       LD     r_3, 0(r_2) % Fst projection
10      LD     r_2, 4(r_2) % Snd projection
11      ADD    r_2, r_3, r_2 % Add projections
12 L12  ADD    r_1, r_2, r_1 % Add integer
13      BR     L2 % Back edge
14 L14  MOV    r_0, r_1 % Copy result to r_0
15      RET    % Result in r_0
```

Safety Proof (1/5)

The Verification Condition

1. The code producer must prove that the code obeys the TIL representation of data.
2. The consumer must be able to check the foreign code against the safety policy.

$\Rightarrow$ Both the code producer and the code consumer must compute a sufficient condition that ensures compliance with the safety policy.

$\Rightarrow$ The *verification condition* is a predicate[a] whose validity w.r.t. the first-order logic rules and the typing rules is the expected sufficient condition.

[a] $P ::= true | P_1 \wedge P_2 | P_1 \supset P_2 | \forall \mathtt{r}_i.P | e_1 = e_2 | e_1 \neq e_2 | e : \tau$

Safety Proof (2/5)

Computing the Verification Condition

All the loop invariants predicates and the interface of the function (pre- and post-conditions predicates) are required. In the example:

Invariant[a] (corresponds to a logical checkpoint):

2 `L2` `INV` $r_m \vdash r_0 :$ `T list` $\wedge r_m \vdash r1 :$ `int` % `Loop invariant`

Precondition (corresponds to a logical assumption):

0 `sum:` `INV` $r_m \vdash r_0 :$ `T list` % r_0 `is l`

Postcondition (corresponds to a logical conclusion):

$$\mathtt{r}_m \vdash \mathtt{r}_0 : \mathtt{int}$$

[a] in general, conjuntion of all typing predicates for the registers that are live at the invariant point

Safety Proof (3/5)

The Verification Condition Generator

For a vector Π of insns, the symbolic interpreter

$$
\begin{array}{ll}
VC_i = & \\
\quad [\mathtt{r}_s + op/\mathtt{r}_d]\, VC_{i+1} & \textit{if } \Pi_i = \texttt{ADD } \mathtt{r}_s, op, \mathtt{r}_d \\
\quad \mathtt{r}_m \vdash \mathtt{r}_s + n : \mathtt{addr} \wedge [\mathtt{sel}(\mathtt{r}_m, \mathtt{r}_s + n)/\mathtt{r}_d]\, VC_{i+1} & \textit{if } \Pi_i = \texttt{LD } \mathtt{r}_d, n(\mathtt{r}_s) \\
\quad \mathtt{r}_m \vdash \mathtt{r}_d + n : \mathtt{addr} \wedge [\mathtt{upd}(\mathtt{r}_m, \mathtt{r}_d + n, \mathtt{r_s}/\mathtt{r}_m]\, VC_{i+1}) & \textit{if} \Pi_i = \texttt{ST } \mathtt{r}_s, n(\mathtt{r}_d) \\
\quad (\mathtt{r}_s = 0 \supset VC_{i+n+1}) \wedge (\mathtt{r}_s \neq 0 \supset \mathtt{VC}_{i+1}) & \textit{if } \Pi_i = \texttt{BEQ } \mathtt{r}_s, n \\
\quad Post & \textit{if } \Pi_i = \texttt{RET} \\
\quad \mathcal{I} & \textit{if } \Pi_i = \texttt{INV } \mathcal{I}
\end{array}
$$

computes in one pass a vector VC of predicates (for instance, for each load, a safety predicate is emitted).

Safety Proof (4/5)

The Verification Condition

The verification condition encodes the semantics of the program:[a]

$$VC(\Pi, Inv, Post) = \forall \mathtt{r}_i.\ \bigwedge_{i \in Inv} Inv_i \supset\ VC_{i+1}$$

Example: the verification condition predicate has 2 conjuncts

$$\overbrace{\mathtt{r}_m \vdash \mathtt{r}_0 : \mathtt{T\ list}}^{\text{precondition}} \supset \overbrace{(\mathtt{r}_m \vdash \mathtt{r}_0 : \mathtt{T\ list} \wedge \mathtt{r}_m \vdash \mathtt{r}_0 : \mathtt{int})}^{\text{loop invariant}}$$

$$\overbrace{\mathtt{r}_m \vdash \mathtt{r}_0 : \mathtt{T\ list} \wedge \mathtt{r}_m \vdash \mathtt{r}_1 : int}^{\text{loop invariant}} \supset \ldots$$

[a] one conjunct for each path from an invariant (including the precondition) to another invariant (including the postcondition)

Safety Proof (5/5)

Soundness

The fact that the verification condition is a sufficient condition is expressed by the 'Theorem of Soundness of VC-based Certification.'

The verification condition $VC(\Pi, Inv, Post)$ is proved by the code producer and this proof $\triangleright VC(\Pi, Inv, Post)$ is the safety proof.

Encoding of Safety Proofs (1/6)

We would like to have an encoding function K such that:

$$K(P \wedge R) = K(\wedge)\ K(P)\ K(R)$$

$$K\left(\begin{array}{cc} D_1 & D_2 \\ \hline \triangleright P & \triangleright R \\ \hline \multicolumn{2}{c}{\triangleright P \wedge R} \end{array}\right) = K(\wedge_{intro}) K(P)\ K(R)\ K(D_1)\ K(D_2)$$

such that proof-checking can be done efficiently

$\Rightarrow$ Two-stage encoding of the derivation $\triangleright VC(\Pi, Inv, Post)$:

1. Representation of predicates and proofs as objects in LF[a],
encoding through the *representation function*, where proof-checking is done efficiently by the type-checker
2. Encoding of LF objects in a compact binary format.

[a] Edinburgh Logical Framework

Encoding of Safety Proof (2/6)

Signature...

Expressions, predicates, and ML types are declared has Type:

```
exp : Type
pred : Type
tp : Type
```

Proof rules are declared as a familiy of types:

```
pf : pred -> Type
```

A safety proof is a proof of $\triangleright VC(\Pi, Inv, Post)$, i.e. a closed object of type `pf` $K(VC(\Pi, Inv, Post))$.

Encoding of Safety Proofs (3/6)

...Signature...

Expressions and predicates constructors[a], and the type judgment are associated constants:

```
0: exp
>: exp -> exp -> exp
+: exp -> exp -> exp
true: pred
and: pred -> pred -> pred
impl: pred -> pred -> pred
all: (exp->pred)->pred
hastype: exp->exp->tp->pred
```

[a] we recall that $P ::= true | P_1 \wedge P_2 | P_1 \supset P_2 | \forall \mathtt{r}_i.P | e_1 = e_2 | e_1 \neq e_2 | e : \tau$

Encoding of Safety Proofs (4/6)

...Signature

Proof rules are associated constants:

`and_i` : Π `p`:`pred`.Π `r`:`pred`.`pf p` −> `pf r` −> `pf` (**and** `p r`)
`impl_i` : Π `p`:`pred`.Π `r`:`pred`.(`pf p` −> `pf r`)−>`pf` (`impl p r`)
`all_i` : Π `p`:`exp`−>`pred`.(Π `v`:`exp`.`pf` (`p v`))−>`pf` (`all p`)

Type inference rules also [a] :

`tp_list` : Π `m`:`exp`.Π `e`:`exp`.Π `t`:`tp`.
`pf` (`hastype m e` (`list t`)) −> `pf` (`neq e` 0) −>
`pf` (**and** (**and** (`hastype m e addr`)
(`hastype m` (`sel m e`) `t`))

[a] We recall that:

$$\frac{m \vdash e : \tau\texttt{list} \quad e \neq 0}{m \vdash e : \texttt{addr} \wedge m \vdash e + 4 : \texttt{addr} \wedge m \vdash \texttt{sel}(m, e) : \tau \wedge m \vdash \texttt{sel}(m, e + 4) : \tau\ \texttt{list}}$$

```
(and (hastype m (+ e 4) addr)
     (hastype m (sel m (+ e 4)) (list t))))
```

Encoding of Safety Proof (5/6)

Representation Function...

We take the representation function $\ulcorner \cdot \urcorner$ to be our encoding function K.

$$\ulcorner x \urcorner = x$$

$$\ulcorner P \wedge R \urcorner = \texttt{and} \ulcorner P \urcorner \ulcorner R \urcorner$$

$$\frac{\begin{array}{cc} \ulcorner D_1 & D_2 \urcorner \\ \triangleright P & \triangleright R \end{array}}{\triangleright P \wedge R} = \texttt{and_i} \ulcorner P_1 \urcorner \ulcorner P_2 \urcorner \ulcorner D_1 \urcorner \ulcorner D_2 \urcorner$$

Encoding of Safety Proof (6/6)

...Representation Function

$$\ulcorner \forall x.P \urcorner = \texttt{all}(\lambda x : \texttt{exp}.\ulcorner P \urcorner)$$

$$\frac{\begin{array}{c}\ulcorner \vdots D_v \urcorner \\ \triangleright [v/x]P_x\end{array}}{\triangleright \forall x.P_x} = \texttt{all_i}(\lambda x : \texttt{exp}.\ulcorner P_x \urcorner)(\lambda v : \texttt{exp}.\ulcorner D_v \urcorner)$$

$$\frac{\begin{array}{c}\ulcorner \triangleright P \urcorner \\ \vdots D_u \\ \triangleright R\end{array}}{\triangleright P \supset R} = \texttt{impl_i}\ulcorner P \urcorner \ulcorner R \urcorner(\lambda u : \texttt{pf}\ulcorner P \urcorner.\ulcorner D_u \urcorner)$$

Proof-Carrying Code

Conclusion

The search for a closed object of type `pf` $K(VC(\Pi, Inv, Post))$ is automatic.

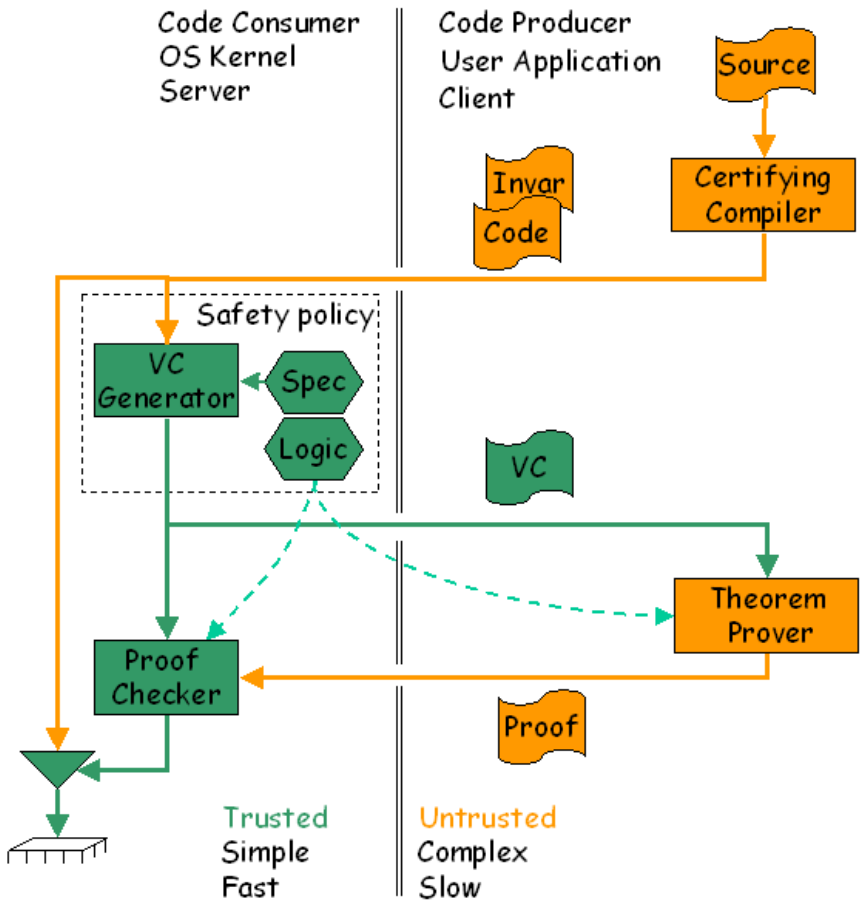
The difficult work of finding loop invariants is done off-line.

Intuition: no scalability, should rather rely on the compiler to produce proofs that the target code preserves interesting properties of the source program.

$\Rightarrow$ *Certifying Compilation*

The Design and Implementation of a Certifying Compiler ([NL98])

Certifying compiler = Compiler + *Certifier*



The Compiler

Source language: C (type safe, bound-checked arrays, nor casts nor pointer arithmetic, no `free`)
Target language: DEC Alpha assembly language

Optimizations:

- registers usage optimizations
- array-bound checks removal

Additional output:

- code annotations: the types of the live registers at the beginning of the loop body (the earlier loop invariants)
- type specification: types of registers containing arguments and results (the earlier pre- and post-conditions)

The Certifier

The certifier checks the type safety and memory safety of assembly-language target program and (a) generate a formal proof of safety or (b) a counter-proof. Three steps:

1. Verification condition generator (as before)
2. Prover: guaranteed to be able to prove the safety predicate automatically
3. Proof checker

Nice side-effects:

- neither the compiler nor the proof checker need to be correct for us to be able to detect incorrect code
- the certifier improves on compiler testing
- the verification condition generator and the checker are simple

Conclusion

Certified compilation and certifying compilation share the same goal (correctness of compilation results) and the same inner machinery (theorem proving).

The difference resides in the definition of correctness: full semantic equivalence for the former, type and memory safety for the later.

Related Work

[CNL00] (a certifying compiler for Java)

References

[Ber98] Yves Bertot. A certified compiler for an imperative language. Technical Report RR-3488, INRIA, Sep 1998.

[Ber01] Yves Bertot. Formalizing a JVML verifier for initialization in a theorem prover. In *Proceedings of CAV'01*, volume 2102 of *Lecture Note in Computer Science*, pages 14–24. Springer Verlag, 2001.

[BF95] Yves Bertot and Ranan Fraer. Reasoning with executable specifications. In *International Joint Conference of Theory and Practice of Software Development (TAPSOFT/FASE'95)*, volume 915 of *Lecture Notes in Computer Science*. Springer-Verlag, May 1995.

[Bou95] Samuel Boutin. Preuve de correction de la compilation

de mini-ml en code cam dans le système d'aide à la démonstration coq. Technical Report 2536, INRIA, Apr 1995.

[CNL00] Christopher Colby, George C. Necula, and Peter Lee. A certifying compiler for java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000.

[Hat95] John Hatcliff. Mechanically verifying the correctness of an offline partial evaluator. In *Proceedings of the Seventh International Symposium on Programming Languages, Implementations, Logics and Programs*, volume 982 of *Lecture Notes in Computer Science*, pages 279–298. Springer-Verlag, Sep 1995.

[Nec97] George C. Necula. Proof-carrying code. In *Symposium on Principles of Programming Languages*. ACM

SIGPLAN-SIGACT, 1997.

[NL98] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1998.

[PMW01] Christine Paulin-Mohring and Benjamin Werner. Calcul des constructions inductives. Notes de cours, Paris VI, 2001.